.NET Full Stack Development Program

Day9 T-SQL Query

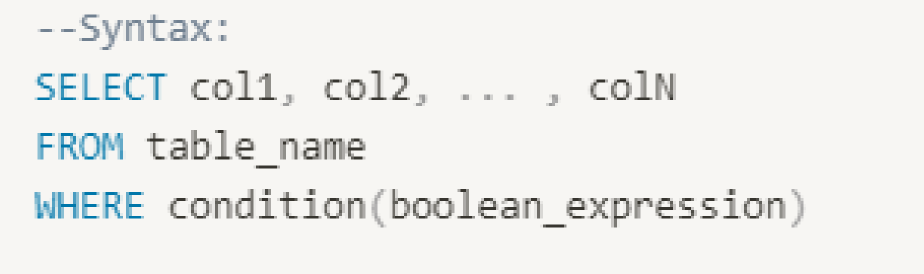
Outline

- Basic SQL Query
- Aggregate Function
- Join
- Sub-query & Set Operator
- View
- Variable
- Procedure
- Index

Select From Where

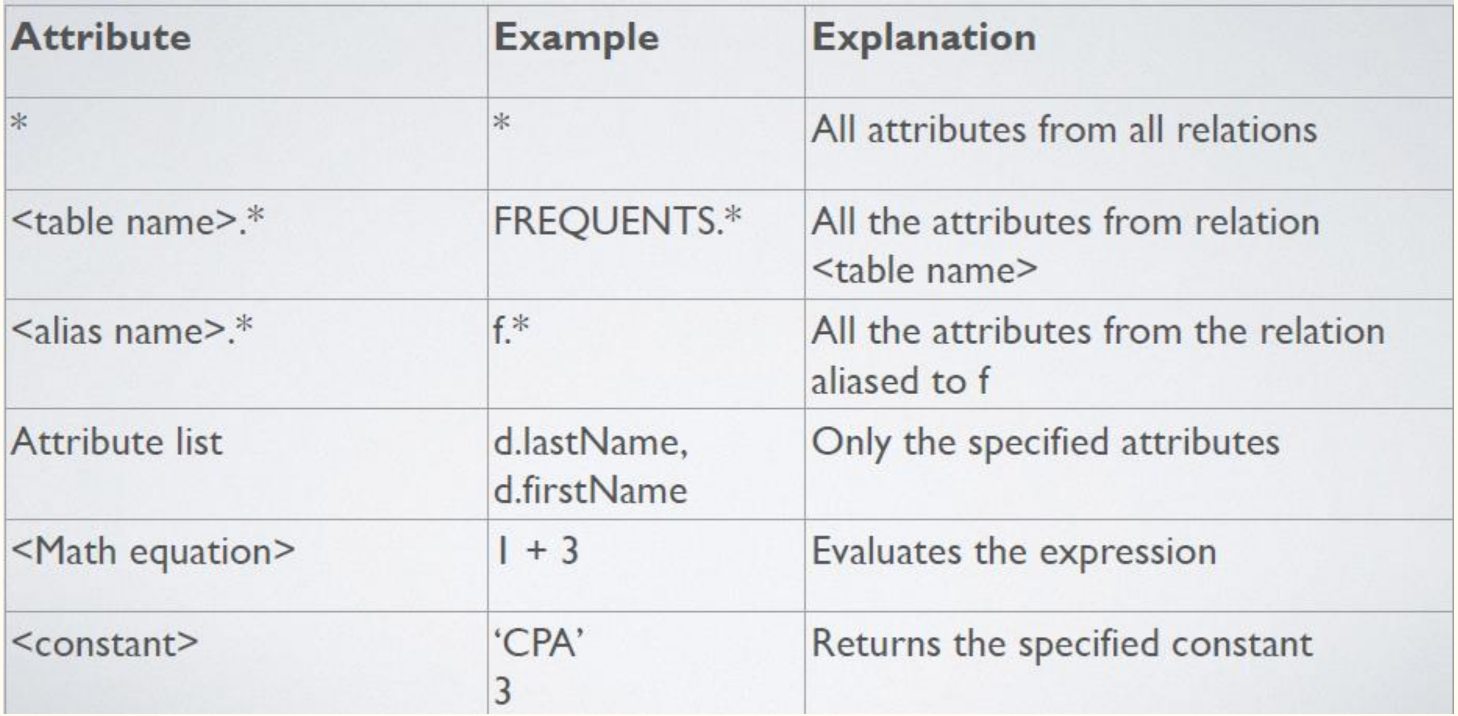
- The **SELECT** statement is used to select data from a database.
- The **WHERE** clause is used to filter records
- The data returned is stored in a result table, called the result set.

- Example:



```
--Syntax:
SELECT col1, col2, ... , colN
FROM table_name
WHERE condition(boolean_expression)
```

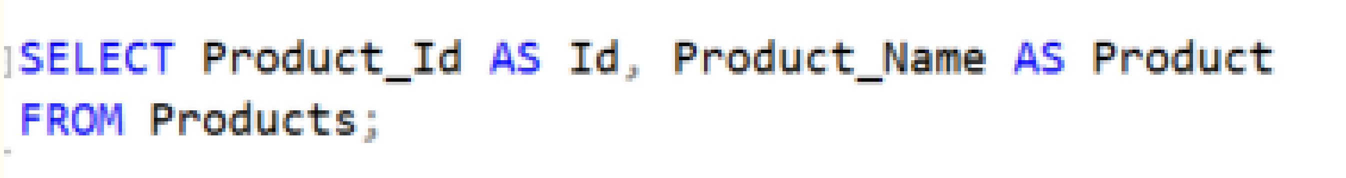
SELECT Clauses



Attribute	Example	Explanation
*	*	All attributes from all relations
<table name>.*	FREQUENTS.*	All the attributes from relation <table name>
<alias name>.*	f.*	All the attributes from the relation aliased to f
Attribute list	d.lastName, d.firstName	Only the specified attributes
<Math equation>	1 + 3	Evaluates the expression
<constant>	‘CPA’ 3	Returns the specified constant

AS

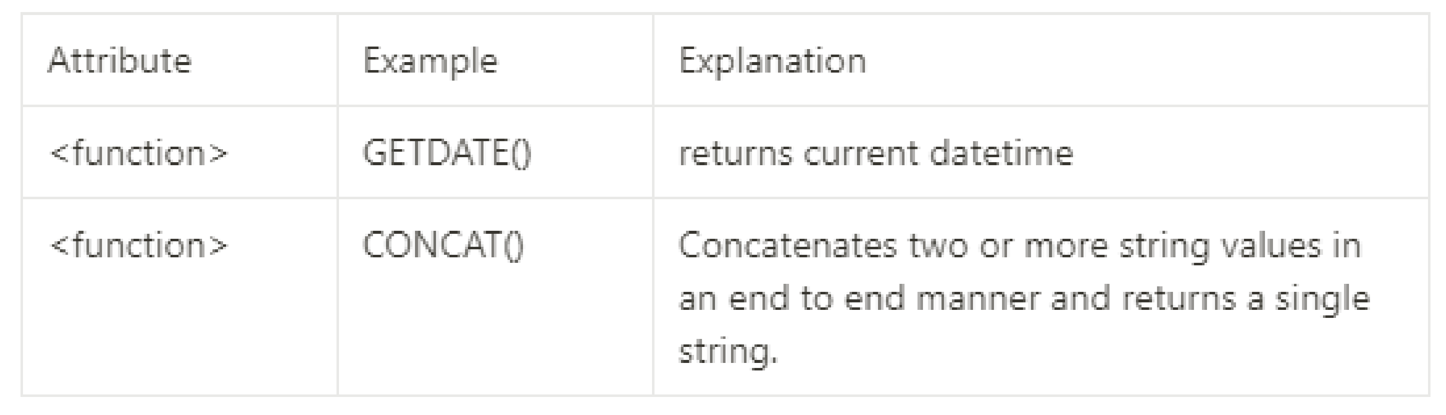
- The AS command is used to create a more meaningful name by renaming a column or table with alias.
- It’s optional and only exists for the duration of the query.



```
SELECT Product_Id AS Id, Product_Name AS Product
FROM Products;
```

More SELECT Clauses

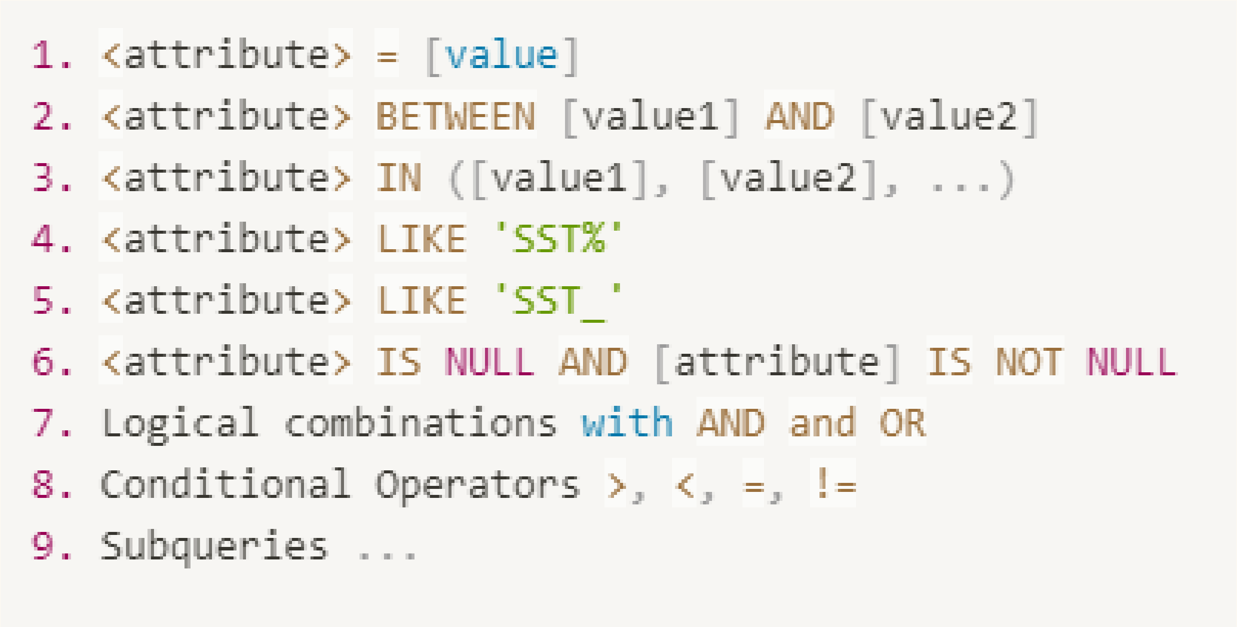
- Functions can be used in SELECT statement



Attribute	Example	Explanation
<function>	GETDATE()	returns current datetime
<function>	CONCAT()	Concatenates two or more string values in an end to end manner and returns a single string.

- More build-in functions:
- https://learn.microsoft.com/en-us/sql/t-sql/functions/functions?view=sql-server-ver15

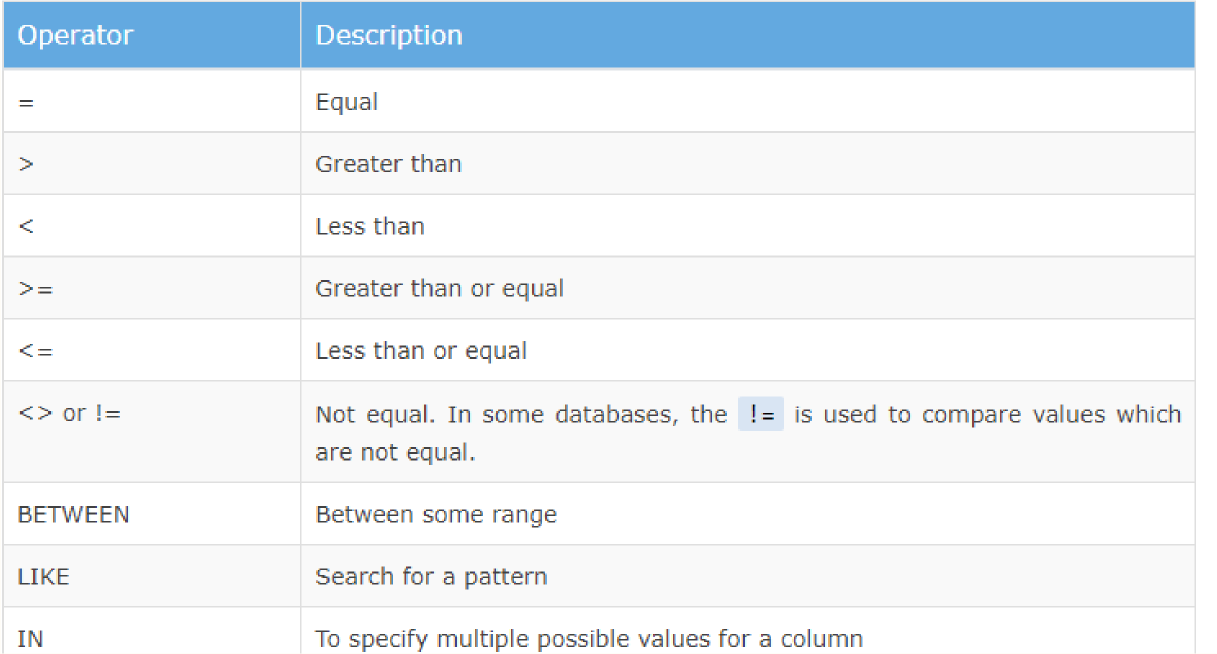
WHERE Clause



```
1. <attribute> = [value]
2. <attribute> BETWEEN [value1] AND [value2]
3. <attribute> IN ([value1], [value2], ...)
4. <attribute> LIKE 'SST%'
5. <attribute> LIKE 'SST_'
6. <attribute> IS NULL AND [attribute] IS NOT NULL
7. Logical combinations with AND and OR
8. Conditional Operators >, <, =, !=
9. Subqueries ...
```

WHERE Clause

- Conditional Operators



Operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<> or !=	Not equal. In some databases, the != is used to compare values which are not equal.
BETWEEN	Between some range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

Aggregate Functions

- Aggregate functions are build-in functions that used to perform simple statistics
  - COUNT()
  - AVG()
  - SUM()
  - MAX()
  - MIN()

- The COUNT() function returns the number of rows that matches a specified criterion.

- The AVG() function returns the average value of a numeric column.

- The SUM() function returns the total sum.

- The MIN() and MAX() functions return the smallest and largest values of the selected columns respectively.

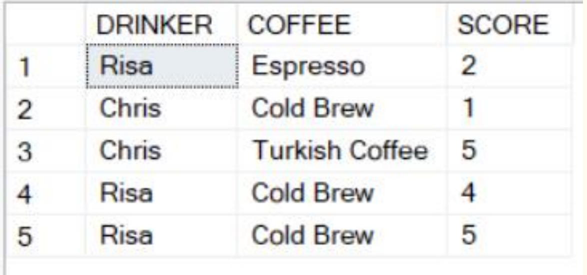
Aggregate Functions

- How does aggregate functions handle Null value?
  - COUNT(*) gives the total number of records in the table including Null values, there is no difference between these two functions (*can be replaced by any number in the int capacity)
  - COUNT(column_name) only considers rows where the column contains a Not-Null value
  - AVG, MIN, MAX, etc. ignore Null values
  - GROUP BY includes a row for null.

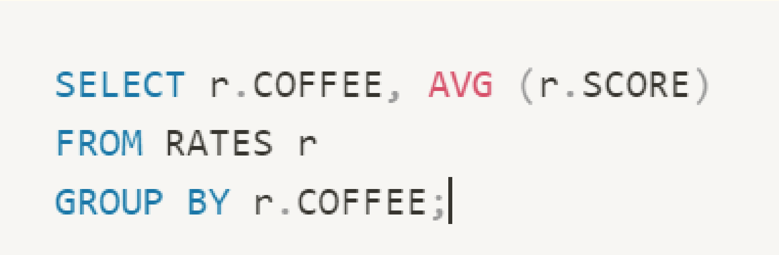
https://learn.microsoft.com/en-us/sql/t-sql/functions/count-transact-sql?view=sql-server-ver16

Group By

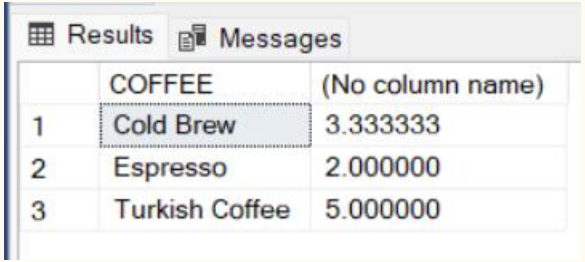
- In SQL Server, the GROUP BY clause is used to from the groups of records
- Any non-aggregate columns called in the select statement must be in group by
- Optional



	DRINKER	COFFEE	SCORE
1	Risa	Espresso	2
2	Chris	Cold Brew	1
3	Chris	Turkish Coffee	5
4	Risa	Cold Brew	4
5	Risa	Cold Brew	5



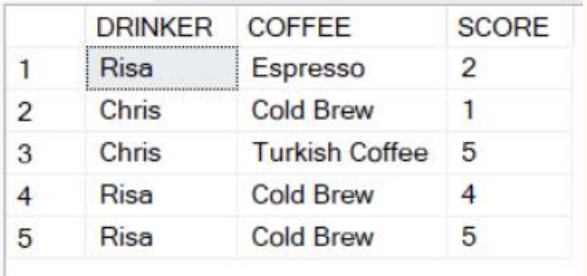
```sql
SELECT r.COFFEE, AVG (r.SCORE)
FROM RATES r
GROUP BY r.COFFEE;
```



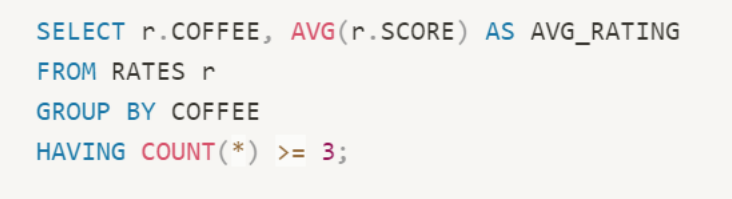
	COFFEE	(No column name)
1	Cold Brew	3.333333
2	Espresso	2.000000
3	Turkish Coffee	5.000000

Having

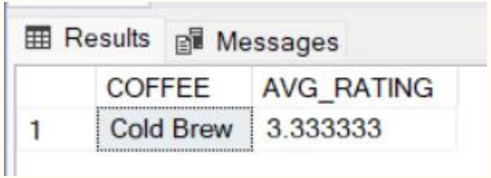
- Having clause is used to filter out grouping records, it’s like the WHERE clause. The difference is WHERE clause cannot be used with aggregate functions, whereas Having clause can work with
- Having clause must come after GROUP BY clause and before ORDER BY clause
- Optional



	DRINKER	COFFEE	SCORE
1	Risa	Espresso	2
2	Chris	Cold Brew	1
3	Chris	Turkish Coffee	5
4	Risa	Cold Brew	4
5	Risa	Cold Brew	5



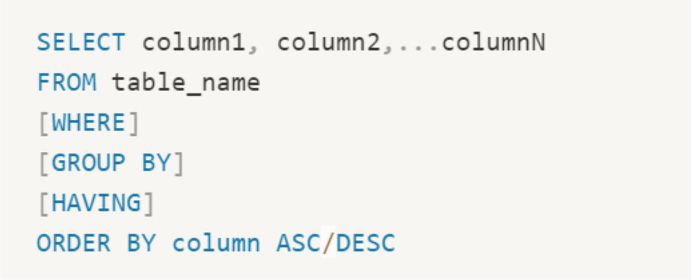
```sql
SELECT r.COFFEE, AVG(r.SCORE) AS AVG_RATING
FROM RATES r
GROUP BY COFFEE
HAVING COUNT(*) >= 3;
```



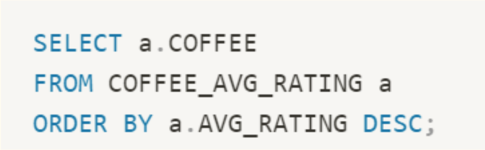
	COFFEE	AVG_RATING
1	Cold Brew	3.333333

Order By

- The ORDER BY clause is used to sort the result in ascending or descending order.
- ORDER BY clause sorts the result in ascending order by default. To sort the result by descending order, use the DESC keyword.
- The ORDER BY must come after WHERE, GROUPBY, and HAVING clause if present in the query.



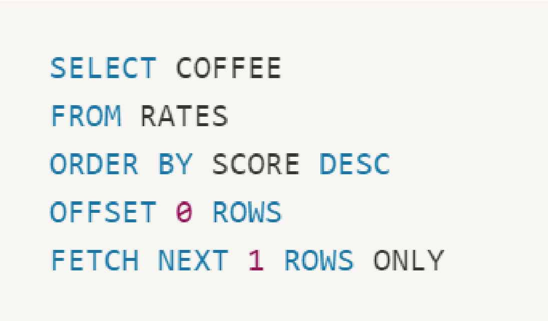
```
SELECT column1, column2,...columnN
FROM table_name
[WHERE]
[GROUP BY]
[HAVING]
ORDER BY column ASC/DESC
```



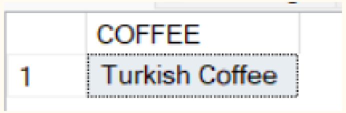
```
SELECT a.COFFEE
FROM COFFEE_AVG_RATING a
ORDER BY a.AVG_RATING DESC;
```

Offset & Fetch

- The ***OFFSET*** clause specifies the number of rows to skip before starting to return rows from the query.
- The offset_row_count can be a constant, variable, or parameter that is greater or equal to zero.
- The ***FETCH*** clause specifies the number of rows to return after the OFFSET clause has been processed.
- The offset_row_count can be a constant or a variable that is greater or equal to one.



```
SELECT COFFEE
FROM RATES
ORDER BY SCORE DESC
OFFSET 0 ROWS
FETCH NEXT 1 ROWS ONLY
```

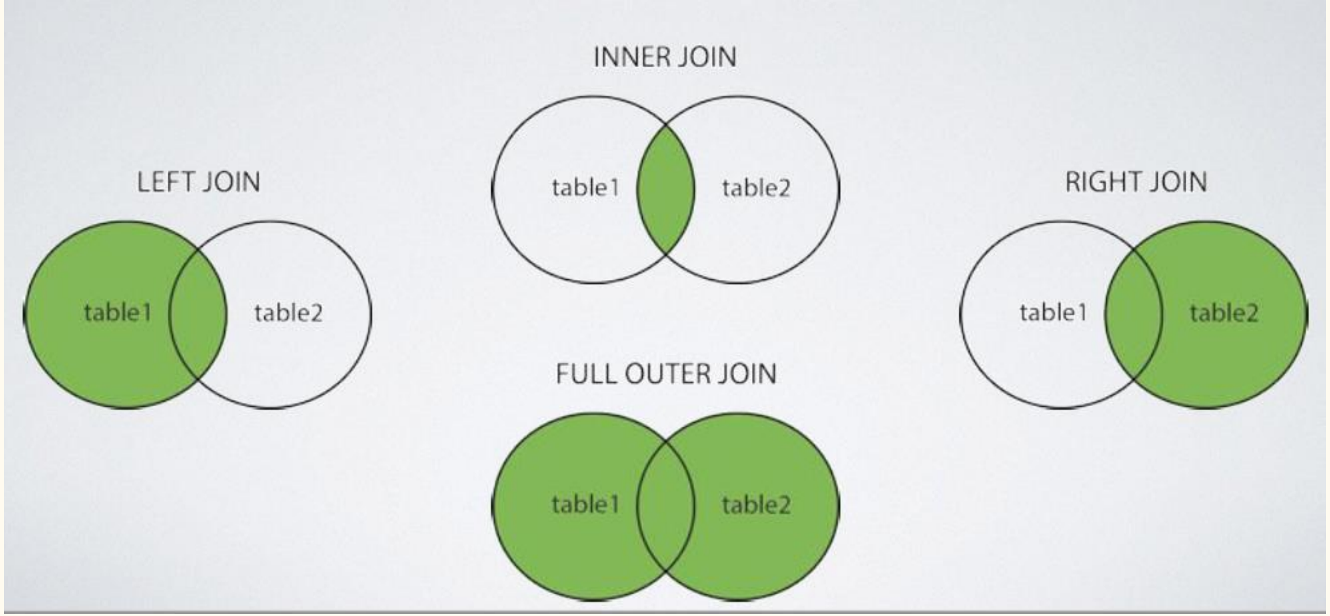


	COFFEE
1	Turkish Coffee

Joins

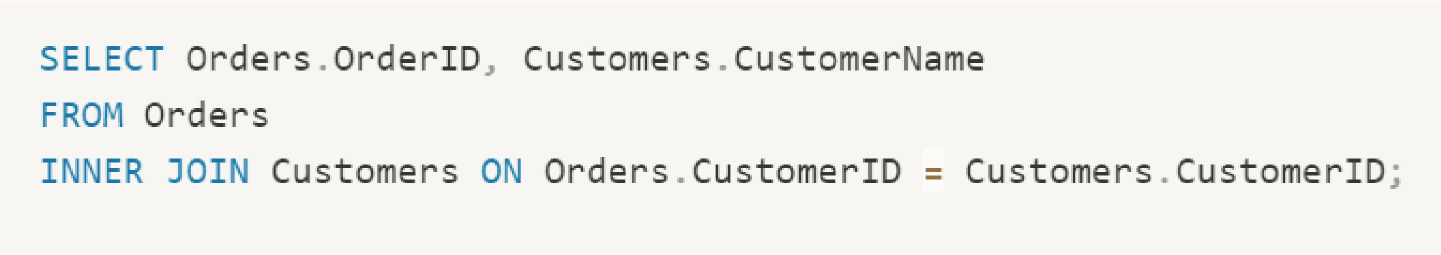
- A JOIN clause is used to combine rows from 2 or more tables, based on a matching column.
- Uses matching data in specified columns to combine or sort data.
- Columns DO NOT have to have the same name.
- Columns DO NOT need to be keys.
- Scope: table to table, table to view, table to synonyms.

Joins


INNER JOIN
table1
table2
LEFT JOIN
table1
table2
RIGHT JOIN
table1
table2
FULL OUTER JOIN
table1
table2

Inner Join

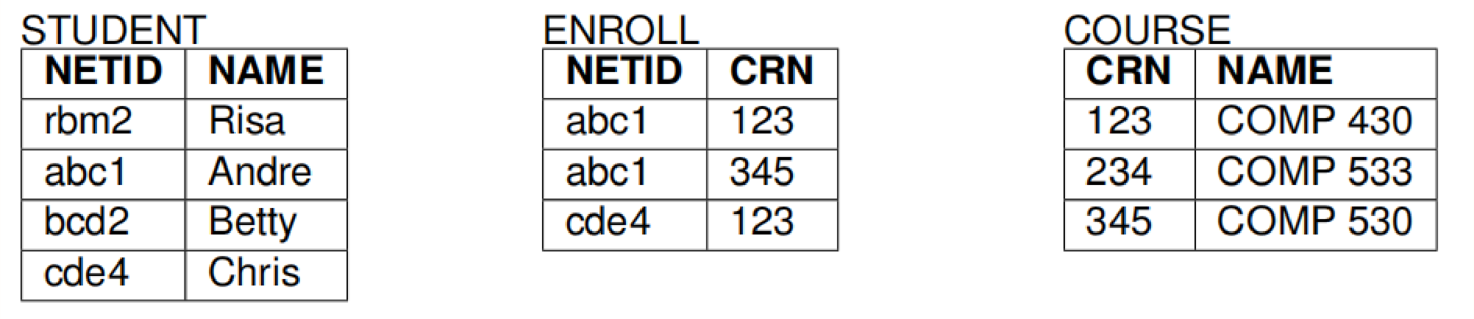
- Inner Join returns records that have matching values in both tables.
- Inner Join selects all rows from both tables as long as there is a match between the columns. If there are records in one table that do not have matches in the other table these records will not be shown.



```
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

Inner Join

- Query all the students who enroll in the courses and list their enrolled courses



STUDENT

NETID	NAME
rbm2	Risa
abc1	Andre
bcd2	Betty
cde4	Chris

ENROLL

NETID	CRN
abc1	123
abc1	345
cde4	123

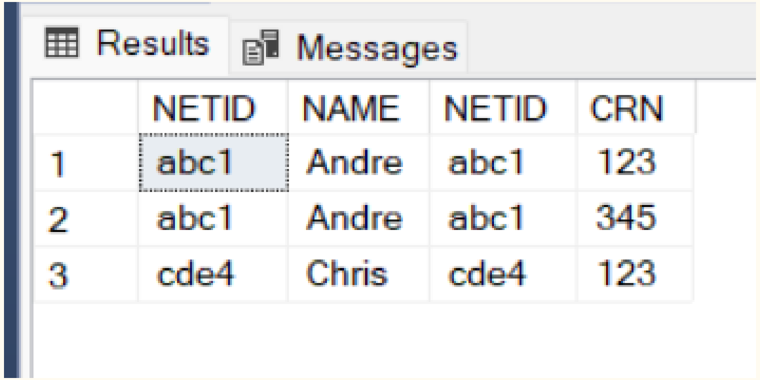
COURSE

CRN	NAME
123	COMP 430
234	COMP 533
345	COMP 530

Inner Join

- Query:

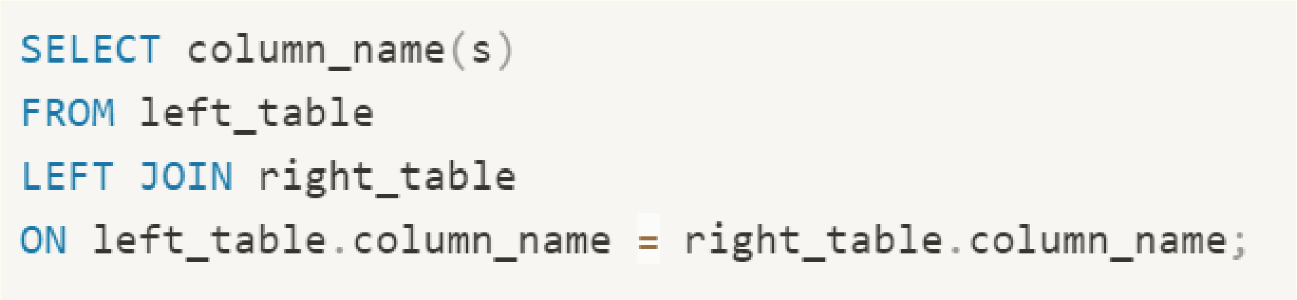
```
SELECT *
FROM STUDENT s
INNER JOIN ENROLL e ON s.NETID = e.NETID
```



	NETID	NAME	NETID	CRN
1	abc1	Andre	abc1	123
2	abc1	Andre	abc1	345
3	cde4	Chris	cde4	123

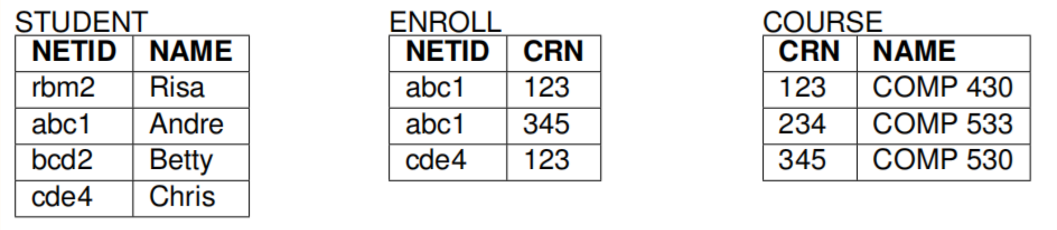
Left/Right Join

- The LEFT JOIN keyword returns all records from the left table and the matching records from the right table.
- If there is **no matching record** for the left table, **NULL will be assigned in the result set.**
- It’s a good idea to choose one direction (either LEFT or RIGHT) and use it to maintain consistency.



```
SELECT column_name(s)
FROM left_table
LEFT JOIN right_table
ON left_table.column_name = right_table.column_name;
```

Left Join



STUDENT

NETID	NAME
rbm2	Risa
abc1	Andre
bcd2	Betty
cde4	Chris

ENROLL

NETID	CRN
abc1	123
abc1	345
cde4	123

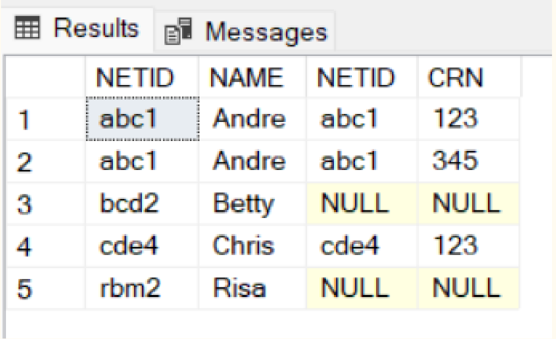
COURSE

CRN	NAME
123	COMP 430
234	COMP 533
345	COMP 530

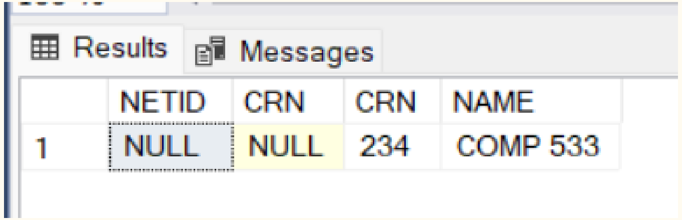
- Query all students and show the enrollment process

```
SELECT *
FROM STUDENT s
LEFT JOIN ENROLL e ON s.NETID = e.NETID
```

```
SELECT *
FROM ENROLL e
RIGHT JOIN COURSE c ON e.CRN = c.CRN
WHERE e.CRN IS NULL
```

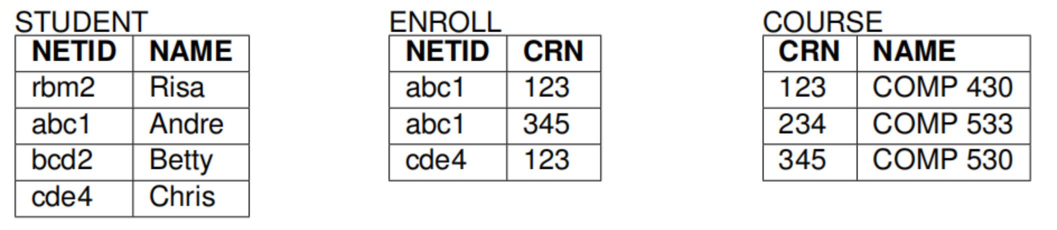


	NETID	NAME	NETID	CRN
1	abc1	Andre	abc1	123
2	abc1	Andre	abc1	345
3	bcd2	Betty	NULL	NULL
4	cde4	Chris	cde4	123
5	rbm2	Risa	NULL	NULL



	NETID	CRN	CRN	NAME
1	NULL	NULL	234	COMP 533

Right Join



STUDENT

NETID	NAME
rbm2	Risa
abc1	Andre
bcd2	Betty
cde4	Chris

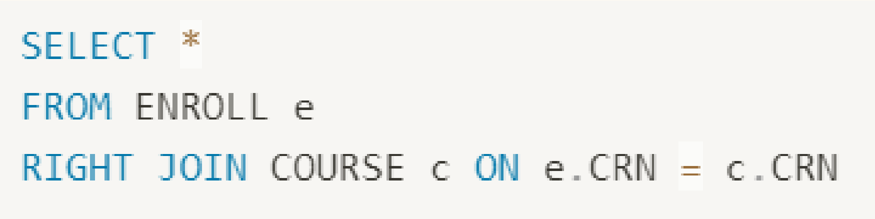
ENROLL

NETID	CRN
abc1	123
abc1	345
cde4	123

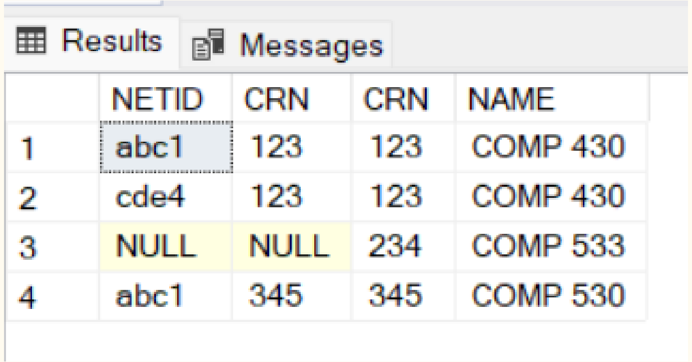
COURSE

CRN	NAME
123	COMP 430
234	COMP 533
345	COMP 530

- Query all the enrollment status in course

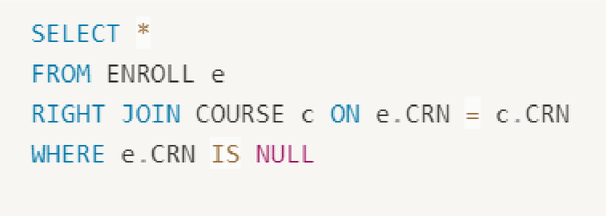


```
SELECT *
FROM ENROLL e
RIGHT JOIN COURSE c ON e.CRN = c.CRN
```

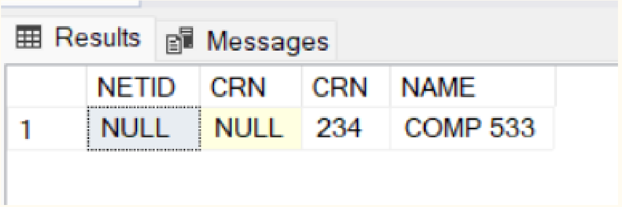


Results Messages

	NETID	CRN	CRN	NAME
1	abc1	123	123	COMP 430
2	cde4	123	123	COMP 430
3	NULL	NULL	234	COMP 533
4	abc1	345	345	COMP 530



```
SELECT *
FROM ENROLL e
RIGHT JOIN COURSE c ON e.CRN = c.CRN
WHERE e.CRN IS NULL
```

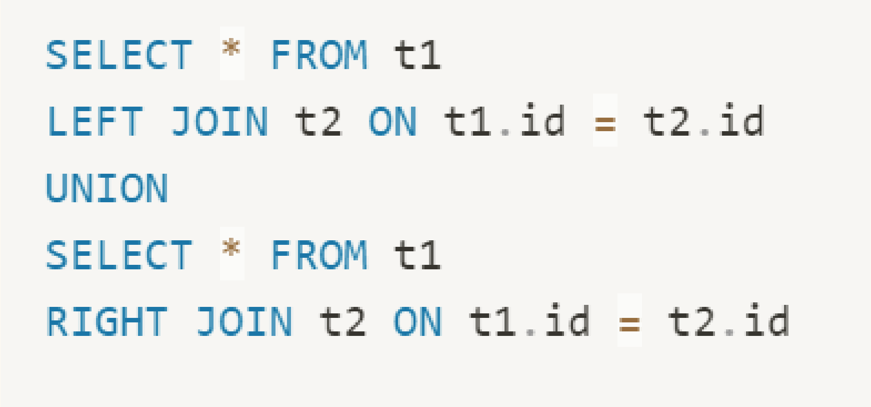


Results Messages

	NETID	CRN	CRN	NAME
1	NULL	NULL	234	COMP 533

Full Outer Join

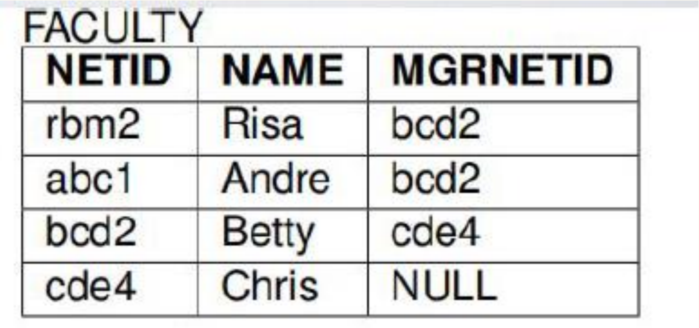
- Used to match up tuples from different relations
- Includes all the relations from both sides
- If there is no matching tuple, shows NULL



```
SELECT * FROM t1
LEFT JOIN t2 ON t1.id = t2.id
UNION
SELECT * FROM t1
RIGHT JOIN t2 ON t1.id = t2.id
```

Self Join

- SELF JOIN is used when a JOIN is used on the same table.



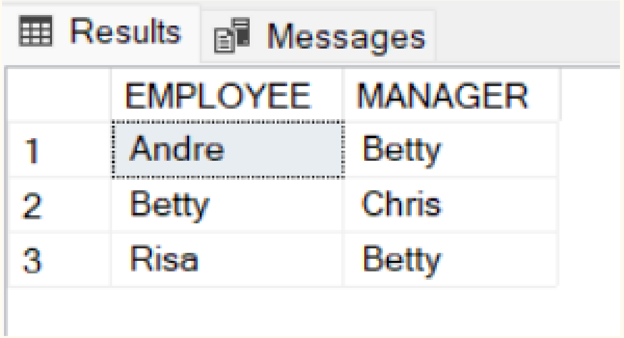
FACULTY

NETID	NAME	MGRNETID
rbm2	Risa	bcd2
abc1	Andre	bcd2
bcd2	Betty	cde4
cde4	Chris	NULL

Self Join

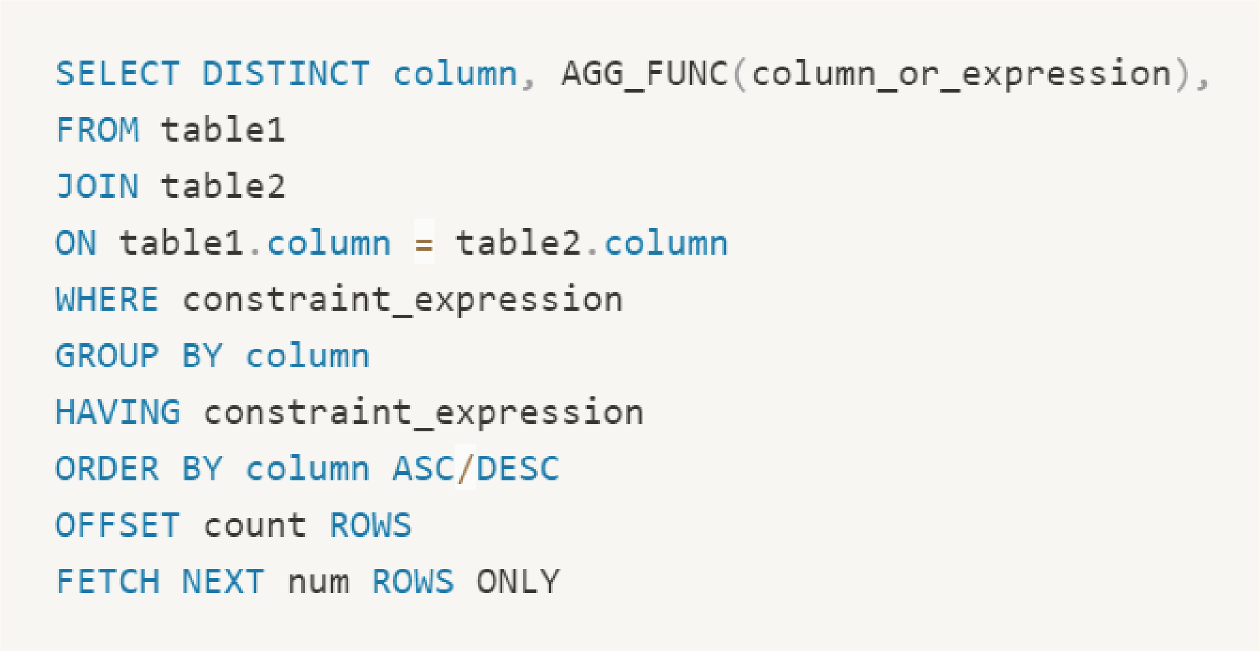
- Query

```
SELECT F.NAME AS EMPLOYEE, MGR.NAME AS MANAGER
FROM FACULTY F JOIN FACULTY MGR
ON F.MGRNETID = MGR.NETID
```



	EMPLOYEE	MANAGER
1	Andre	Betty
2	Betty	Chris
3	Risa	Betty

Complete Query Example



```sql
SELECT DISTINCT column, AGG_FUNC(column_or_expression),
FROM table1
JOIN table2
ON table1.column = table2.column
WHERE constraint_expression
GROUP BY column
HAVING constraint_expression
ORDER BY column ASC/DESC
OFFSET count ROWS
FETCH NEXT num ROWS ONLY
```

Set Operations

- Results are unordered. It could be useful to perform operations on these:

  Union

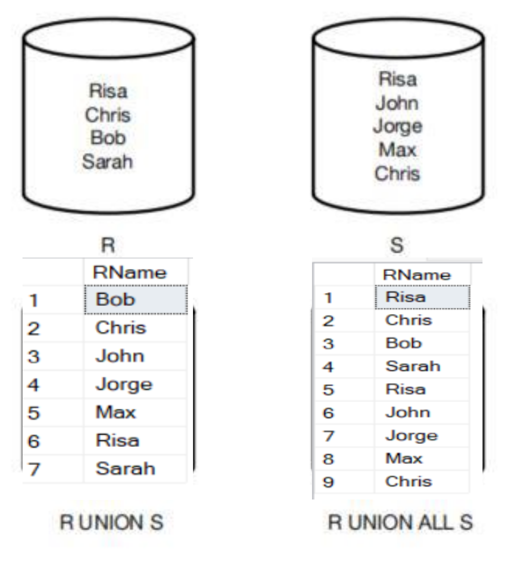
  Intersection

  Difference

- Different RDBMS provide different levels of support

UNION and UNION ALL

- UNION- eliminates duplicates
- UNION ALL- does NOT eliminate duplicates
- Uses the column names from the first result set
- *Data types* must match
- *Number of attributes* must match



	RName
1	Bob
2	Chris
3	John
4	Jorge
5	Max
6	Risa
7	Sarah

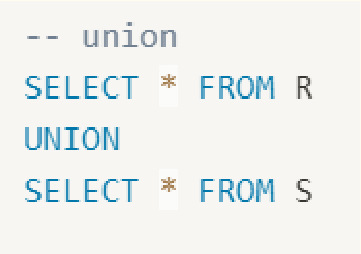
R UNION S

	RName
1	Risa
2	Chris
3	Bob
4	Sarah
5	Risa
6	John
7	Jorge
8	Max
9	Chris

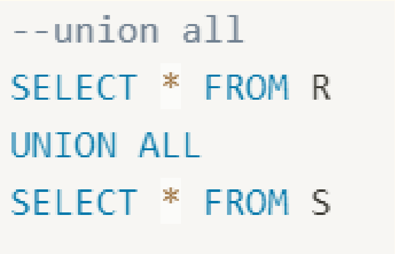
R UNION ALL S

UNION and UNION ALL Example

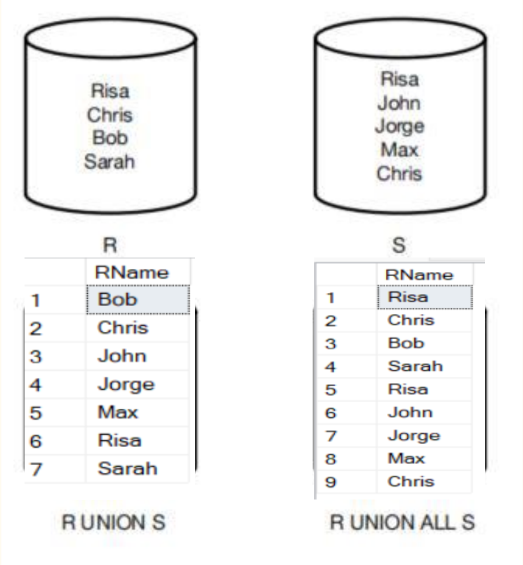
- R(RName)
- S(SName)



```
-- union
SELECT * FROM R
UNION
SELECT * FROM S
```



```
--union all
SELECT * FROM R
UNION ALL
SELECT * FROM S
```



	RName
1	Bob
2	Chris
3	John
4	Jorge
5	Max
6	Risa
7	Sarah

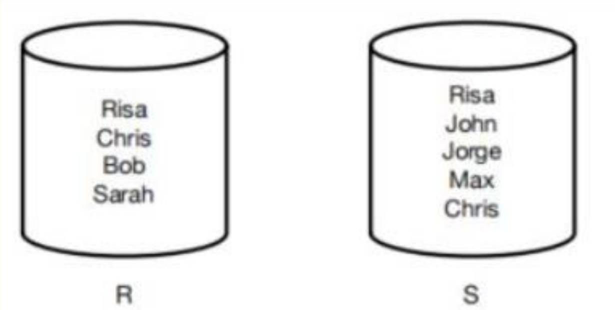
R UNION S

	RName
1	Risa
2	Chris
3	Bob
4	Sarah
5	Risa
6	John
7	Jorge
8	Max
9	Chris

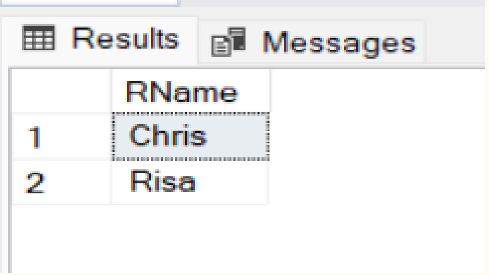
R UNION ALL S

Intersection

- Intersection Implemented via ***INTERSECT***



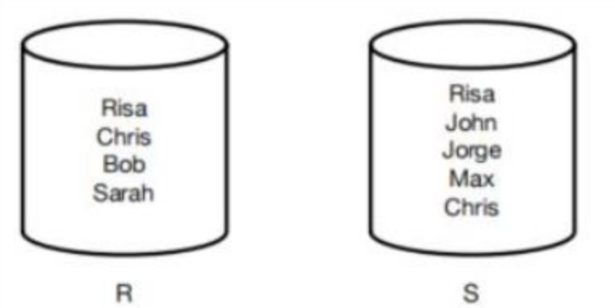
```
SELECT * FROM R
INTERSECT
SELECT * FROM S
```

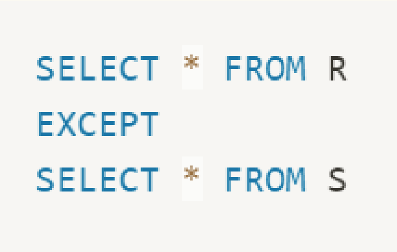


	RName
1	Chris
2	Risa

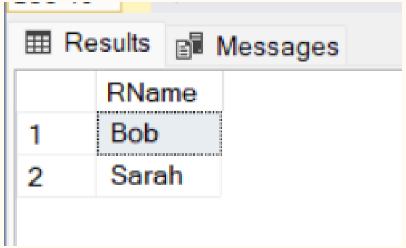
Difference

- Difference Implemented via ***EXCEPT***
  - Display the values in the first select statement MINUS any values found in the second select statement





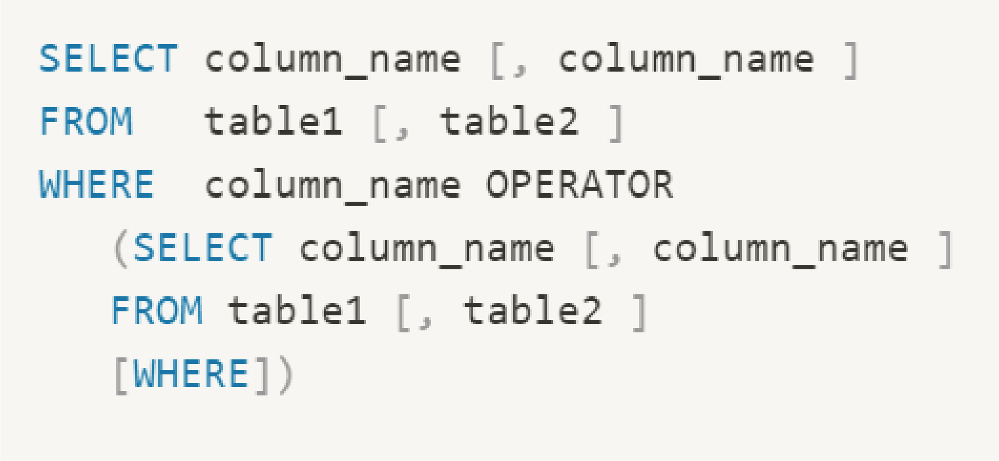
```
SELECT * FROM R
EXCEPT
SELECT * FROM S
```



	RName
1	Bob
2	Sarah

Subquery

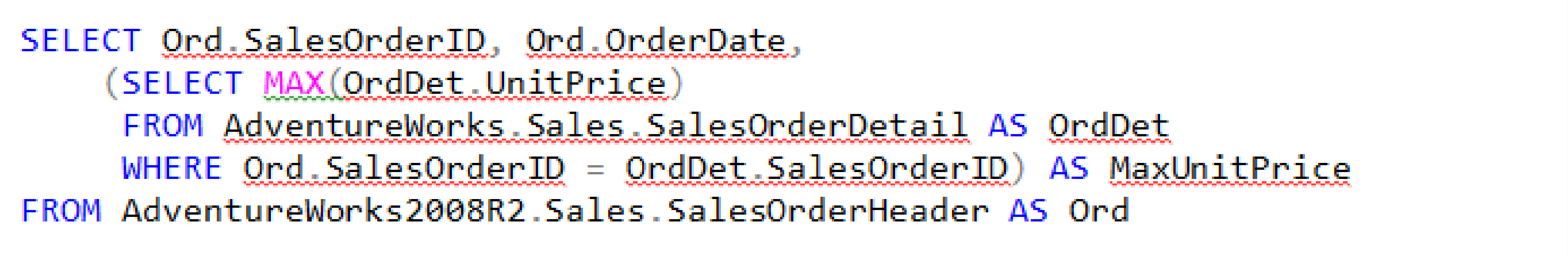
- A subquery is a query that is nested inside a SELECT, INSERT, UPDATE, DELETE statement or inside another subquery.
- Subqueries must be written in parentheses.
- Subqueries must include the SELECT clause and the FROM clause.



```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
   (SELECT column_name [, column_name ]
   FROM table1 [, table2 ]
   [WHERE])
```

Where to Put a Subquery

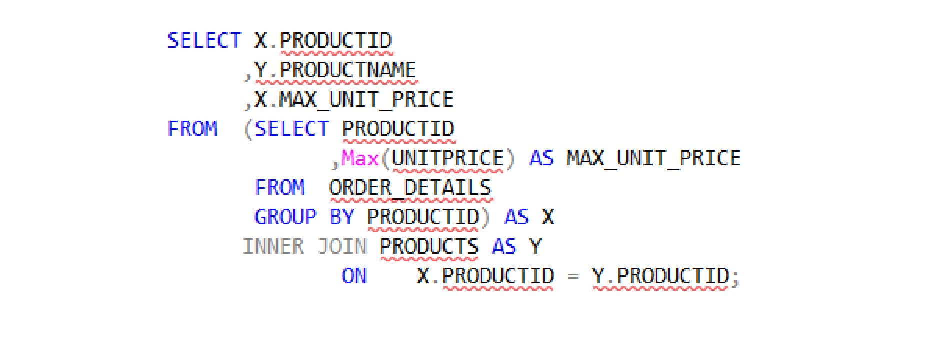
- SELECT clause
    - Can be used to retrieve values in a select clause, but only if they return a single result



```sql
SELECT Ord.SalesOrderID, Ord.OrderDate,
    (SELECT MAX(OrdDet.UnitPrice)
     FROM AdventureWorks.Sales.SalesOrderDetail AS OrdDet
     WHERE Ord.SalesOrderID = OrdDet.SalesOrderID) AS MaxUnitPrice
FROM AdventureWorks2008R2.Sales.SalesOrderHeader AS Ord
```

Where to Put a Subquery

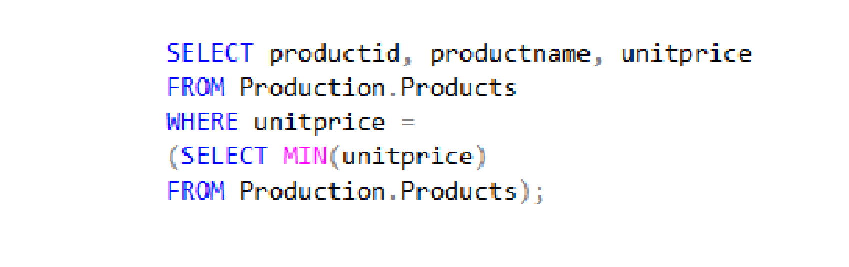
- FROM clause
  - Can be used to return an entire table but **must have an alias**
  - Derived Table



```
SELECT X.PRODUCTID
      ,Y.PRODUCTNAME
      ,X.MAX_UNIT_PRICE
FROM  (SELECT PRODUCTID
             ,Max(UNITPRICE) AS MAX_UNIT_PRICE
       FROM  ORDER_DETAILS
       GROUP BY PRODUCTID) AS X
      INNER JOIN PRODUCTS AS Y
              ON     X.PRODUCTID = Y.PRODUCTID;
```

Where to Put a Subquery

- WHERE clause
  - Most common use
  - Used to filter results based on another table



```
SELECT productid, productname, unitprice
FROM Production.Products
WHERE unitprice =
(SELECT MIN(unitprice)
FROM Production.Products);
```

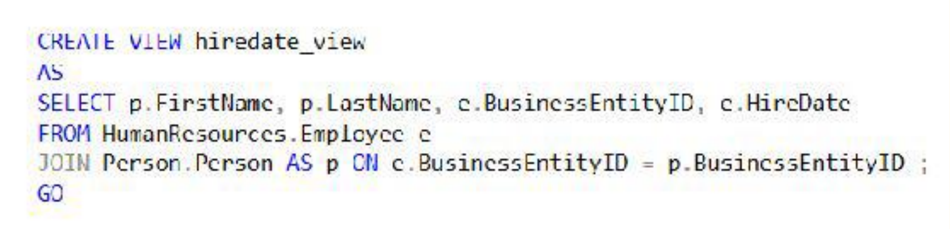
View

- A view is often seen as a **virtual table**

- It displays data that you choose, but does not actually hold any data

- Good for security since you can prevent showing extra data

- DML operations just happen on the table.
  - you can modify data on view level, and the source data will be updated as well.

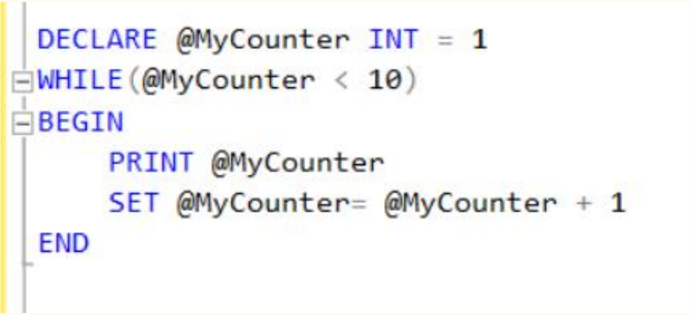


```
CREATE VIEW hiredate_view
AS
SELECT p.FirstName, p.LastName, e.BusinessEntityID, e.HireDate
FROM HumanResources.Employee e
JOIN Person.Person AS p ON e.BusinessEntityID = p.BusinessEntityID ;
GO
```

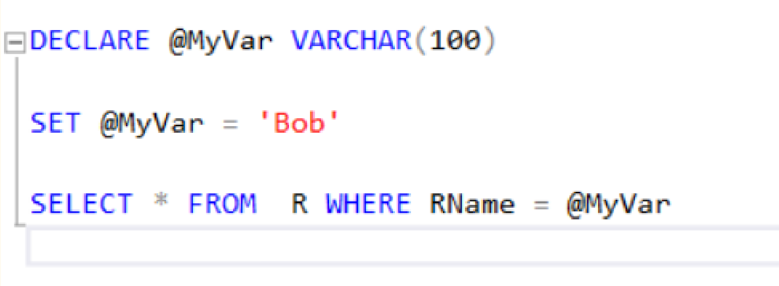
Variable

- A Transact-SQL local variable is an object that can hold a single data value of a specific type. Variables in *batches* and *scripts* are typically used:
    - As a counter either to count the number of times a loop is performed or to control how many times the loop is performed.
    - To hold a data value to be tested by a control-of-flow statement.
    - To save a data value to be returned by a ***stored procedure*** return code or function return value.
- User-Defined Variables are displayed with an "@" symbol
- System Variables are displayed with an "@@" symbol

Variable Example



```
DECLARE @MyCounter INT = 1
WHILE(@MyCounter < 10)
BEGIN
    PRINT @MyCounter
    SET @MyCounter= @MyCounter + 1
END
```



```
DECLARE @MyVar VARCHAR(100)

SET @MyVar = 'Bob'

SELECT * FROM  R WHERE RName = @MyVar
```

Stored Procedure

- In SQL Server, a stored procedure is a set of T-SQL statements that are compiled and stored in the database. The stored procedure accepts input and output parameters, executes the SQL statements, and returns a result set if any.

- System procedures: System procedures are included with SQL Server and are physically stored in the internal, hidden Resource database and logically appear in the sys schema of all the databases. The system-stored procedures start with the sp_ prefix.

https://learn.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/system-stored-procedures-transact-sql?view=sql-server-ver16

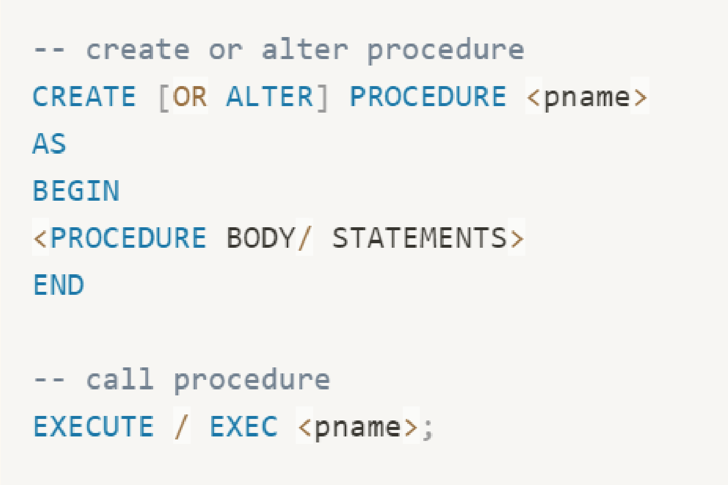
User Defined Stored Procedure

- User Defined Stored Procedures are just Stored Procedures, but created by the user

- Contains statements including calling other stored procedures

- Can have different Input and Output Parameters

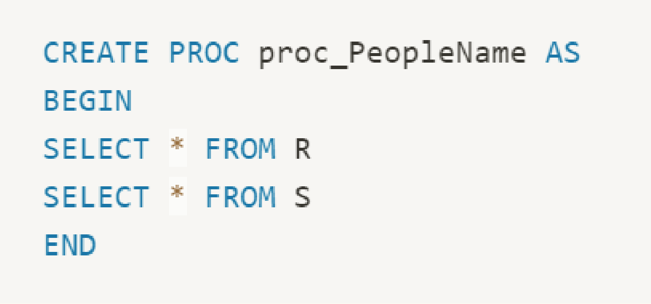
- Must be recompiled after time or changes



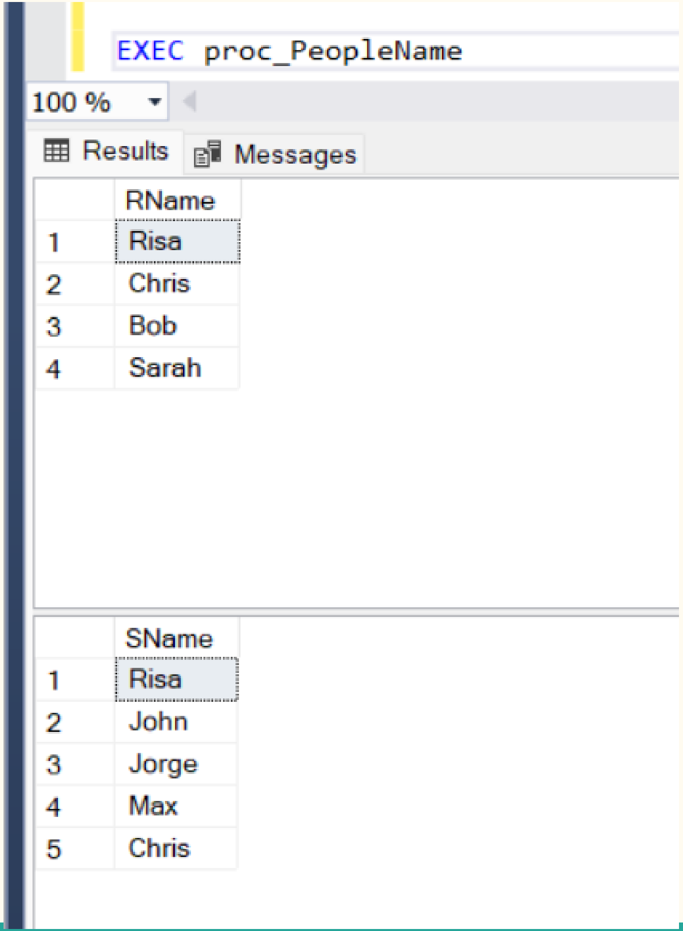
```
-- create or alter procedure
CREATE [OR ALTER] PROCEDURE <pname>
AS
BEGIN
<PROCEDURE BODY/ STATEMENTS>
END

-- call procedure
EXECUTE / EXEC <pname>;
```

User Defined Stored Procedure

- (basic)
- A stored procedure can have zero or more INPUT and OUTPUT parameters.



```
CREATE PROC proc_PeopleName AS
BEGIN
SELECT * FROM R
SELECT * FROM S
END
```



```
EXEC proc_PeopleName
```

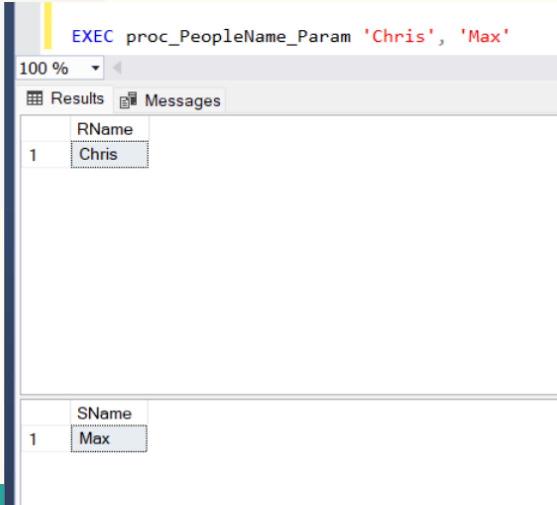
	RName
1	Risa
2	Chris
3	Bob
4	Sarah

	SName
1	Risa
2	John
3	Jorge
4	Max
5	Chris

User Defined Stored Procedure

- (SP w/ input parameters)
- Each parameter is assigned a name, and a data type, if no other statement follows, then this parameter is treated as an INPUT parameter

```
CREATE PROC proc_PeopleName_Param(@Name1 VARCHAR(20), @Name2 VARCHAR(20)) AS
BEGIN
SELECT * FROM R WHERE RName = @Name1
SELECT * FROM S WHERE SName = @Name2
END
```



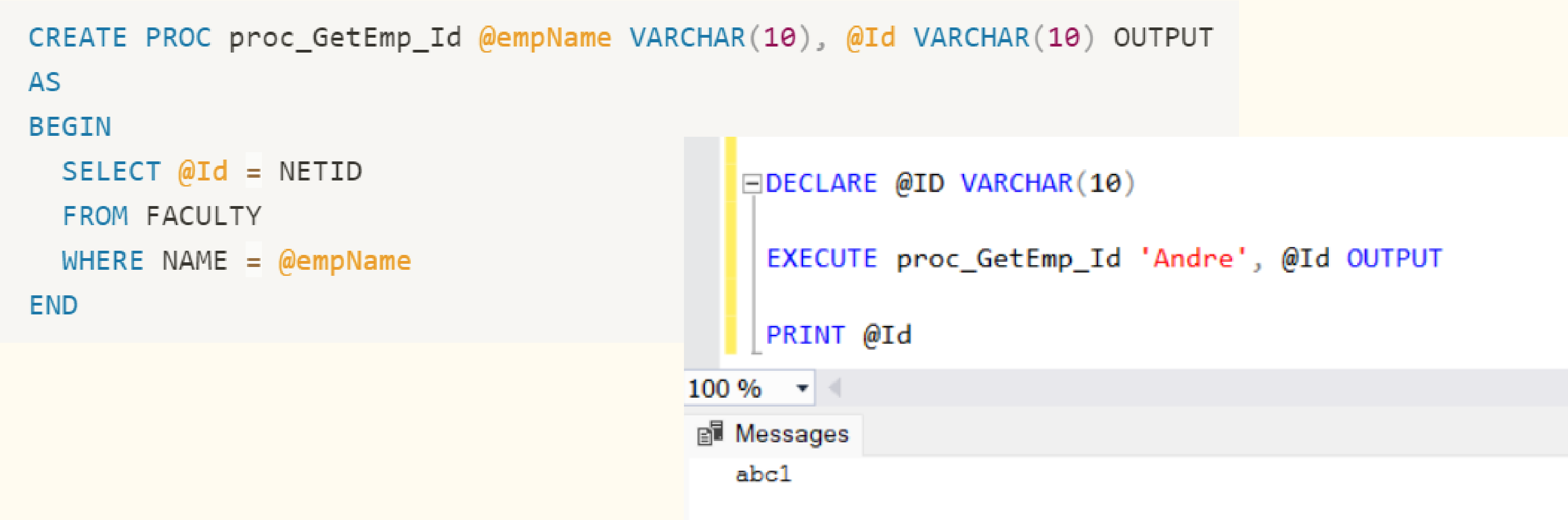
```
EXEC proc_PeopleName_Param 'Chris', 'Max'
```

	RName
1	Chris

	SName
1	Max

User Defined Stored Procedure

- (SP w/ both INPUT and OUTPUT)
- Stored procedures can return a value to the calling program if the parameter is specified as OUTPUT.



```sql
CREATE PROC proc_GetEmp_Id @empName VARCHAR(10), @Id VARCHAR(10) OUTPUT
AS
BEGIN
  SELECT @Id = NETID
  FROM FACULTY
  WHERE NAME = @empName
END
```

```sql
DECLARE @ID VARCHAR(10)

EXECUTE proc_GetEmp_Id 'Andre', @Id OUTPUT

PRINT @Id
```

```
abc1
```

Query Execution

Step 1:

- Parser check query syntax
- Break query to token --> (intermediate files)

Step 2:

- Query Optimizer creates the best possible execution plan based on current resource utilization

Step 3:

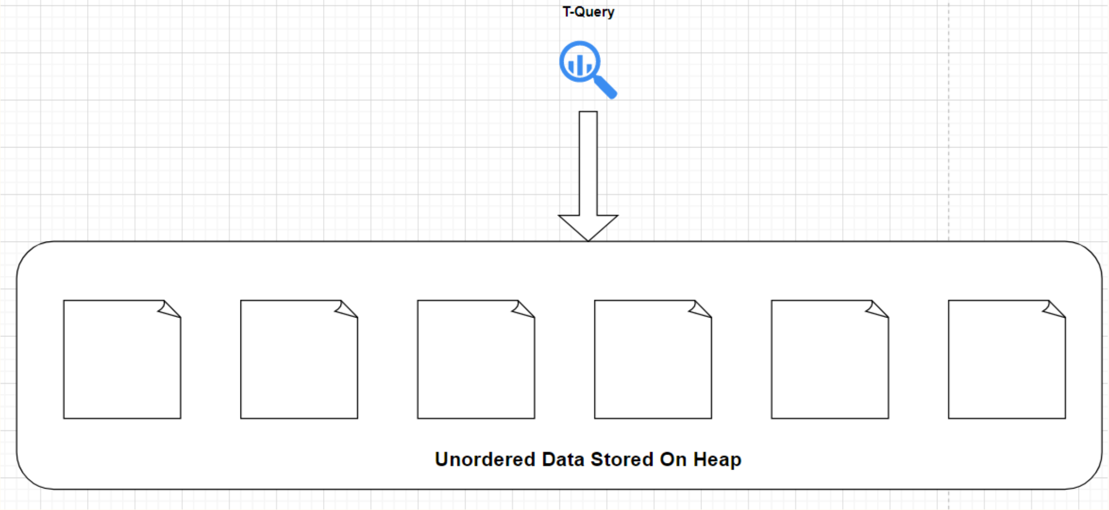
- DB engine --> Run the query

Index

- It's used to sort and **optimize data fetch time**
- Operate similar to index in a book
- When created, an index will create a dynamic Balance Tree (B+ Tree)
- Primary key creates Clustered Index, unique key creates Non-Clustered Index
- Tables without a Clustered Index are called **HEAP** Tables
- Keys ≠ Indexes

Index

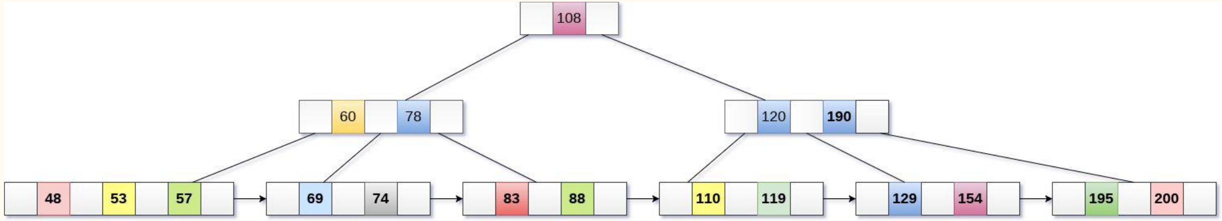
- If a table has no index at all, when new data is inserted, they are added wherever there is free space, and in no particular order.



CLUSTERED INDEX

- Composed of 3 main levels
  - Root Level
  - Intermediate Level
  - Leaf Page Level

- Each Node is about 8KB in size
  - 8060B for data
  - 132B for pointers
  - 8192B in Total



Each Index created will have a Balance tree structure to be used, but the type of Index will determine how data is stored in a Balance Tree

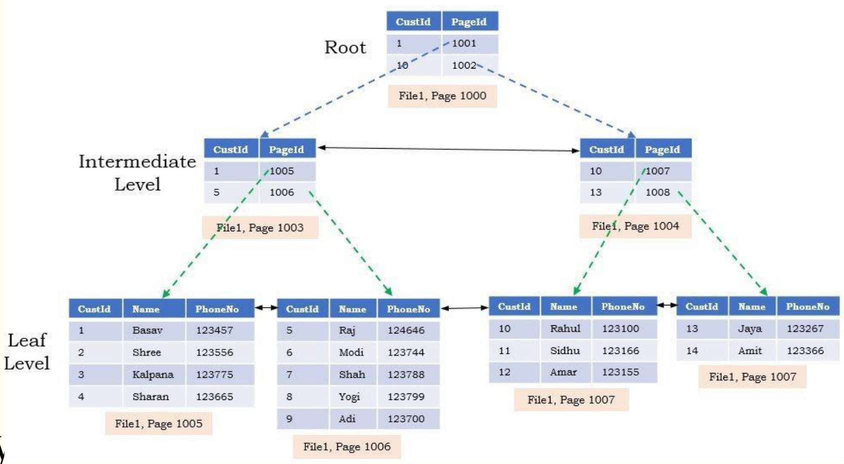
Clustered Indexes will store data in Leaf Pages and sort them based on the Key values of the column you choose.

Non-Clustered Indexes will **NOT** store data in the Leaf Pages, instead they'll point to the rows they're referencing

CLUSTERED INDEX

- A clustered index will physically move the data from the table into it’s Balance Tree
- The data is now matching physically and logically
- Data is sorted based on ascending order for the column chosen, this becomes the clustering key
- This is why there can only be 1 Clustered
- Index on a table, data can only be physically sorted and stored once

B+ Tree Structure of a Clustered Index



CustId	PageId
1	1001
10	1002

File1, Page 1000

CustId	PageId
1	1005
5	1006

File1, Page 1003

CustId	PageId
10	1007
13	1008

File1, Page 1004

CustId	Name	PhoneNo
1	Basav	123457
2	Shree	123556
3	Kalpana	123775
4	Sharan	123665

File1, Page 1005

CustId	Name	PhoneNo
5	Raj	124646
6	Modi	123744
7	Shah	123788
8	Yogi	123799
9	Adi	123700

File1, Page 1006

CustId	Name	PhoneNo
10	Rahul	123100
11	Sidhu	123166
12	Amar	123155

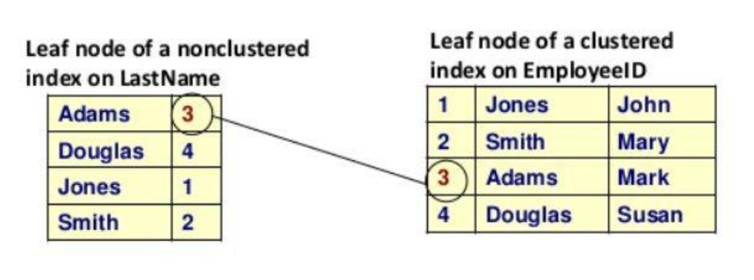
File1, Page 1007

CustId	Name	PhoneNo
13	Jaya	123267
14	Amit	123366

File1, Page 1007

NON-CLUSTERED INDEX

- Since Non-Clustered Indexes do not physically move or store data, there can be many on a single table.
    - Currently up to 999 different Indexes
- A Non-Clustered Index on a table with a Clustered Index must now grab data from the B-Tree of the CI.
- So data will come up through the Root of the CI and fall into the Leaf Pages of the NCI



Leaf node of a nonclustered index on LastName

Adams	3
Douglas	4
Jones	1
Smith	2

Leaf node of a clustered index on EmployeeID

1	Jones	John
2	Smith	Mary
3	Adams	Mark
4	Douglas	Susan

Questions

- How does Index improve the performance?

  Find Index vs. Table Scan

- Will Index always improve the performance?

  Maintain the index

Questions?